**Kamila Szewczyk**

# KamilaLisp

## THE COMPLETE REFERENCE

2021 - 2023

## Abstract

KamilaLisp is a Lisp-inspired, dynamically typed programming language that borrows many ideas from Haskell, Standard ML, APL, Scheme and others. It is a functional programming language with a strong emphasis on array programming, and it is designed to be used in a wide range of applications. Its design stems from previous iterations, which featured more orthodox Lisp syntax, lazy evaluation and symbolic computation with an emphasis on mathematical programming.

# Foreword

KamilaLisp as a language has its origins in its previous iterations, v0.1 and MalbolgeLISP. The first iteration of MalbolgeLISP (v1.0) was released in August 2020 and had very few of the features that distinguish KamilaLisp today. The second iteration of MalbolgeLISP (v1.1) was released in July 2021 and was the subject of much attention due to its unusual choice of implementation language. The final MalbolgeLISP (v1.2) was released in September 2021, sharing many ideas and features with KamilaLisp. Unfortunately, the effort required to implement it continued to grow. The codebase was becoming increasingly disorganised and difficult to maintain. In addition, the implementation of new complicated features was increasingly difficult due to the choice of implementation language. The first version of KamilaLisp, a MalbolgeLISP-inspired Lisp dialect, appeared in December 2021, featuring lazy evaluation, sophisticated numerical and symbolic operations, and many original design choices borrowed from MalbolgeLISP. Progress on the language has stalled due to significant performance problems with the Java Virtual Machine, the overhead of lazy evaluation, and some questionable decisions made at the core of the interpreter. After realising this rather early, an effort to create a C++ rewrite of the original code base began towards the end of 2021, but was quickly abandoned due to problems with garbage collection, efficient memory management, and C++ lacking certain qualities that would otherwise make it a good language in which to implement an interpreter for KamilaLisp. In August 2022 I started working on KamilaLisp v0.2, a Java rewrite of the original codebase that does not have some of the problematic features of the previous versions. This book, in the hope of being useful to the determined reader who wants to learn about KamilaLisp, describes my experiences in implementing a complex language runtime using Java, and also describes the language itself. It is assumed that the reader has an elementary knowledge of functional programming or of the APL programming language.

# Contents

# Glossary

In this book, the following assumptions are made:

- *Folding* a list `ω` with a function `αα` is equivalent to putting `αα` between every element of `ω`, also given the identity element of `αα`. One could assume binding to the left (`(ω[1] αα ω[2]) αα ω[3]`), or to the right (`ω[1] αα (ω[2] αα ω[3])`), hence the names *fold-right* and *fold-left*. Unlike reduction, the result of *folding* an empty array is defined and equal to the identity element.

- *Scanning* a list `ω` with a function `αα` is equivalent to mapping a fold over prefixes of a list. This definition of a scan has a $O(n^2)$ complexity regardless of the fold type. A more efficient, $O(n)$ definition of a *scan-left* exists (fold with intermediate steps).

- *Outer product* of function `αα` (`∘.αα`) and arrays `α` and `ω` is the application of `αα` between every pair of elements from `α` and `ω`. The resulting list has a depth given by `ρ,⍤ρ`.

- *Inner product* of functions `αα`, `ωω` and lists `α` and `ω` (`αα.ωω`) is equivalent to folding the list obtained by putting `ωω` between corresponding pairs of `α` and `ω` with `αα`.

- A *higher-order function* is a function that takes another function as its argument.

- A *lambda expression* (in the context of KamilaLisp) is an anonymous function with static (lexical) scoping, meaning that it can see all the variables bound by its lexical ancestor, unlike to dynamic scoping implemented by older LISPs where a function can see all the variables bound by its caller(s). In both cases, the first bound variable found is used, allowing to shadow variables.

- A *dyad* is a two argument function.

- A *monad* is generally a single argument function. The book also uses it in a context of an abstract data constructor implementing the *bind* and *unit* functions.

- *Replicating* a list `ω` according to list `α` is copying each element of `ω` a given number of times (specified by the corresponding element of `α` or assumed to be zero). In KamilaLisp, `α` can be a scalar, in which case the contents of `ω` are catenated to each other `α` times. If `ω` is also a scalar, it's repeated `α` times to form a list.

- *Filtering* a list `ω` with function `αα` is equivalent to mapping `αα` on every element of `ω` (assuming `αα` returns a boolean value, i.e. `0` or `1`), and replicating `ω` with the result of the mapping (`ω/⍤αα⍤ω`).

- *Rotating* a list `ω` by `α` elements is equivalent to dropping `α` elements from `ω`, and joining them with `ω` (dyadic `ϕ`, or more illustratively, `(↓,↑)`).

- *Zipping* two arrays is the act of forming a list of pairs via the juxtaposition of corresponding elements from the given arrays.

- *Flattening* a list if the act of decreasing the list's depth by one level, enlisting all of the elements from the topmost sublists into a resultant list.

- *Mapping* a function `αα` over a list `ω` is the act of processing each element of `ω` with the function `αα` to produce a resultant list of equivalent length.

- *Partial application* is defined as fixing a number of arguments to a function, yielding an anonymous function of smaller arity.
- *Iteration* of a function over an argument is defined as evaluating the function over its result starting with the initial argument until a condition is satisfied. Sometimes the condition is numeric, or is defined as a dyadic function between the previous and current result (just `while`). A fixed point combinator could be implemented using partial application of deep equality (`⍣≡`). Iteration is faster and consumes less resources than recursion, assuming no tail call optimisation.
- *Function composition* (or `@`/`∘`, as KamilaLisp calls it) is taking an arbitrary amount of functions where each function operates on the results of the function before, except the last function, which gets all the arguments passed to the anonymous function yielded by composition.
- *Selfie* is a higher order function that duplicates the argument to the function it takes (e.g. `×⍨`, computes the square of its argument).
- *Commute* swaps the order of two arguments to a function. `α f ω ⟺ ω f⍨ α`.

# Chapter 1

# Initial considerations

This chapter discusses the basics of KamilaLisp. Throughout this book, the author will use the KamilaLisp interpreter to check and execute the declarations of a program one by one. The emphasis will be put on list processing and mathematical functions to form elementary understanding of the language.

## 1.1. Programs and variables

A KamilaLisp program is a sequence of declarations, which are executed in the order they are written. The first program presented in this book is shown below:

```
--> def a (+ (* 2 3) 2)
8
--> def b (* 5 a)
40
```

It consists of two declarations. The first declaration binds the identifier a to the integer 8, and the second declaration binds the identifier b to the integer 40, which follows from the intuitive understanding of the arithmetic operations. To the reader not accustomed with Lisp-like syntax, every element of the syntax tree that would otherwise be implicitly grouped by a language with usual arithmetical precedence rules is explicitly grouped by parentheses to form a list. The resulting values of variables can be determined as follows:

```
--> ?a
8
--> ?b
40
```

The question mark is a sign for the KamilaLisp interpreter not to evaluate the entire input as an expression, but rather, to query the value of what it refers to.

Every list besides the empty list (usually written as `` `()' `` or alternatively `nil`) has a *head* defined as the first element of it. When a Lisp program is evaluated, the *head* of the current list is assumed to be a callable value, while the rest of the list (also called the *tail*) is assumed to be a list of arguments.

Because the list of arguments to a function (the *tail*) is evaluated before its applied to the *head*, a perceptive reader could point out a potential issue - *How to introduce list literals in the code*? This question is indeed well-founded, since the list literal would be evaluated in order to pass it parameter to some callable object, hence the tail of the literal would be applied to its head, thus behaving undesirably and almost certainly raising an error. Every Lisp dialect addresses this issue in the same way using the quoting mechanism. Simply put, the quote prevents a list from being evaluated. To observe this behaviour, introduce two more functions called `car` and `cdr` to obtain respectively the *head* and *tail* of a list:

```
--> car '(1 2 3)
1
```

```
--> cdr '(1 2 3)
(2 3)
```

KamilaLisp follows scoping rules familiar from other programming languages, such as Scheme or C++ - *static scoping* (also called *lexical scoping*), where an attempt is initially made to resolve a variable in the current scope. If this approach fails, the variable is resolved in the scope of its lexical ancestors until either the interpreter finds an environment where the variable is bound, or raises an error regarding an unbound variable. Additionally, variables may be *shadowed*, as demonstrated below:

```
--> def my-list '(1 2 3)
(1 2 3)
--> car my-list
1
--> def my-list (cdr my-list)
(2 3)
--> car my-list
2
```

However, it is not possible to shadow pre-defined variables and functions in the global scope:

```
--> def car 5
RuntimeException thrown in thread 1dbd16a6:
        def can not shadow or redefine built-in bindings.
    at entity def  1:1
    at def primitive function
```

To fully exercise lexical scoping, the language needs to provide a way of binding names inside a specific block of code (unlike *def* which binds names in the global scope). This can be accomplished in a variety of ways, the most straight-forward one being the *let* construct. The *let* construct binds a list of name/value pairs, and evaluates the body of the construct in the context of the newly created environment. The syntax of the *let* construct is demonstrated by the following example:

```
--> def a 5
5
--> def b 6
6
--> + a b
11
--> let ((a 10) (b 15)) (+ a b)
25
```

## 1.2. Functions and lambda expressions

Functions are the core component of KamilaLisp. They are first-class objects, which means that they can be passed as arguments to other functions, returned from functions, and assigned to variables. The syntax of a function declaration is as follows:

```
--> defun square (x) (* x x)
(λ x . (* x x))
```

The function that has just been declared is called `square`, and it takes one argument called `x`. The body of the function is the expression `(* x x)`. The function returns the value of their expression, which is simply the square of the argument. Notice that when defining a monadic function, a pair of parentheses can be omitted for brevity:

```
--> defun square x (* x x)
(λ x . (* x x))
```

Since `square` is now bound in the global scope, it can be applied to an argument. The code below binds the result of application of the number 5 to the function `square` to the variable `a`:

```
--> def a (square 5)
25
```

Functions do not need to be named. They can be introduced in the code *anonymously* using the *lambda* construct, which opens up many new possibilities. For example, the declaration of the function `square` can be rewritten as follows:

```
--> def square (lambda x (* x x))
(λ x . (* x x))
```

Furthermore, multivariate lambda expressions could serve as a replacement for the *let* construct:

```
--> let ((a 10) (b 15)) (+ a b)
25
--> (lambda (a b) (+ a b)) 10 15
25
```

Since functions are first-class in KamilaLisp, it is now also possible to return them from functions and take them as arguments. The following example demonstrates these programming techniques using the *lambda* construct:

```
--> ; Returns a function that adds a given number to its argument.
--> ; The technique demonstrated is often called "currying".
--> defun make-adder (x) (lambda y (+ x y))
(λ x . (λ y . (+ x y)))
--> def add-5 (make-adder 5)
(λ y . (+ 5 y))
--> add-5 10
15
--> ; Returns a string explaining the value of a function at point.
--> defun explain (f x) (str:format "The value of f(x) for x={?x} is {f x}")
(λ f x . (str:format "The value of f(x) for x={?x} is {f x}"))
--> explain square 4.5
The value of f(x) for x=4.5 is 20.25
```

When writing complex functions, it is of particular interest to keep the partial results obtained during the execution of the function. This can be achieved by cascading the *let* construct, or using the *let-seq* construct. The following examples implement a function that raises its first argument to the eigth power:

```
--> defun p8 x (let-seq (
...   (def y (* x x))
...   (def z (* y y))
...   (* z z)))
(λ x . (let-seq ((def y (* x x)) (def z (* y y)) (* z z))))
--> p8 4
65536
```

```
--> defun p8 x (let ((y (* x x))) (let ((z (* y y))) (* z z)))
(λ x . (let ((y (* x x)) (z (* y y))) (* z z)))
--> p8 4
65536
--> defun p8 x (let ((y (* x x)) (z (* y y))) (* z z))
(λ x . (let ((y (* x x)) (z (* y y))) (* z z)))
--> p8 4
65536
```

Notice that despite using `def`, `defun`, etc..., the *let-seq* construct does not create any bindings in the global scope - the bindings are local to the block.

Function calls in KamilaLisp are performed using a *call stack*. The call stack is a data structure inside of the interpreter that keeps track of the functions that are currently being executed. When a function is called, a new *stack frame* is created and pushed onto the stack. When the function returns, the stack frame is popped from the stack - this way, the interpreter knows where to return to after the function yields a value. When an exception happens, the interpreter will present the user with a *stack trace*, which is a list of functions that were executed before the exception was thrown, for example:

```
--> defun f x (/ 1 0) ; Oops! Division by zero!
(λ x . (/ 1 0))
--> defun g x (f x)
(λ x . (f x))
--> defun h x (g x)
(λ x . (g x))
--> h 10
ArithmeticException thrown in thread 5e82df6a:
    Division by zero
at entity h  1:1
at (λ x . (g x))  1:9
at entity g  1:12
at (λ x . (f x))  1:9
at entity f  1:12
at (λ x . (/ 1 ...))  1:9
at entity /  1:12
at / primitive function
```

## 1.3. Conditional expressions and comparisons

The comparison operators in KamilaLisp don't differ significantly from the ones present in other, perhaps more orthodox programming languages (such as C). It is worth noting that equality is checked using the `=` function, inequality is checked for using the `/=` function, while the `<=>` function is the so-called *three way comparison* operator, which returns -1, 0 or 1 respectively if the first argument is less than, equal to, or greater than the second argument.

KamilaLisp provides a number of conditional expressions, which are used to control the flow of execution. The most basic one is the *if* construct, which takes three arguments - a condition, an expression to be evaluated if the condition is true, and an expression to be evaluated if the condition is false. The syntax of the *if* construct is demonstrated by the following example:

```
--> defun my-abs x (if (< x 0) (- x) x)
(λ x . (if (< x 0) (- x) x))
--> my-abs -5
5
--> my-abs 5
```

```
5
--> my-abs 0
0
```

The *if* construct is a special case of the *cond* construct, which takes a list of pairs of conditions and expressions. The first condition that evaluates to true is used to evaluate the corresponding expression. The syntax of the *cond* construct is demonstrated by the following reimplementation of the three-way comparison operator:

```
--> defun compare (x y) (cond ((< x y) -1) ((> x y) 1) (0))
(λ x y . (cond ((< x y) -1) ((> x y) 1) (0)))
--> compare 5 10
-1
--> compare 10 5
1
--> compare 5 5
0
```

## 1.4. Recursive functions

Recursion is a powerful technique extensively used in functional programming. Its role in KamilaLisp is admittedly not as important as in other functional programming languages, since the language provides a number of other often more wieldy techniques for solving the same problems (e.g. using array programming). However, it is still worth mentioning that KamilaLisp provides a number of tools for writing recursive functions.

The following function implements the *factorial* function in a recursive manner:

```
--> defun factorial (n) (if (= n 0) 1 (* n (factorial (- n 1))))
(λ n . (if (= n 0) 1 (* n (factorial (- n 1)))))
--> factorial 5
120
```

Since KamilaLisp supports arbitrary precision integers, the factorial function can be applied to arbitrarily large numbers. However, at some point, this function will *overflow its call stack*, meaning that the number of recursive calls will exceed the maximum call stack size. This is a common problem in recursive functions, and it can be solved by using the *tail recursion* technique. Tail recursion is a special case of recursion, where the recursive call is the last expression in the function body. In this case, the stack frame of the current function can be reused for the recursive call, which means that the stack will not grow indefinitely. To make use of this technique, the factorial function needs to be altered so that the recursive call is the last expression in the function body:

```
--> defun factorial (n) (let-seq (
...   (defun f' (n acc) (if (= n 0) acc (f' (- n 1) (* n acc))))
...   (f' n 1)))
(λ n . (let-seq ((defun f' (n acc) (if (= n 0) acc (f' (- n 1) (* n acc)))) (f' n 1))))
--> factorial 5
120
```

The factorial function defines a helper function that has an accumulator argument, which is used to store the intermediate results of the computation. The helper function is called recursively, and the accumulator argument is updated with the result of the multiplication. The *let-seq* construct is used to define the helper function, so that it is not visible outside the factorial function. There is one more step to make this function tail-recursive: redefine the self-referential call to the helper function:

```
--> defun factorial (n) (let-seq (
...   (defun f' (n acc) (if (= n 0) acc (&0 (- n 1) (* n acc))))
...   (f' n 1)))
(λ n . (let-seq ((defun f' (n acc) (if (= n 0) acc (&0 (- n 1) (* n acc)))) (f' n 1))))
--> factorial 5
120
```

The self-referential call was replaced by `&0`, which in simple terms is a reference to the current function. This is a special case of the `&` operator, which is used to refer to functions by nesting level in the source code, akin to de Bruijn indices. The `&0` operator refers to the current function, `&1` refers to the function (anonymous or named) that is the lexical ancestor the current function, and so on. This way, the factorial function no longer errors when applied to large numbers, since the stack does not grow indefinitely:

```
--> factorial 1000
402387260077093773543702433923003985719374864210714
632543799910429938512398629020592044208486969404800
; [...]
000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
```

Another, closely related function to the factorial function that can be implemented using tail recursion is the power function. The following function implements the power function in a simple recursive manner:

```
--> defun power (x n) (if (= n 0) 1 (* x (power x (- n 1))))
(λ x n . (if (= n 0) 1 (* x (power x (- n 1)))))
--> power 2 10
1024
```

Once again, the problem is that the stack will overflow when the power function is applied to large numbers. To prevent this and speed up the computation, the power function needs to be first transformed so that the recursive call is the last expression in the function body. Consider the following *trace* of the power function applied to the arguments `2` and `4`:

```
--> power 2 4
(* 2 (power 2 3))
(* 2 (* 2 (power 2 2)))
(* 2 (* 2 (* 2 (power 2 1))))
(* 2 (* 2 (* 2 (* 2 (power 2 0)))))
(* 2 (* 2 (* 2 2)))
(* 2 (* 2 4))
(* 2 8)
16
```

After transforming the function to use an accumulator:

```
--> defun power (x n) (let-seq (
...   (defun f' (x n acc) (if (= n 0) acc (f' x (- n 1) (* x acc))))
...   (f' x n 1)))
(λ x n . (let-seq ((defun f' (x n acc) (if (= n 0) acc (f' x (- n 1) (* x acc)))) (f' x n 1))))
```

When this implementation of the power function is applied to the same arguments as before, the following trace is produced:

```
--> power 2 4
(f' 2 4 1)
(f' 2 3 2)
(f' 2 2 4)
(f' 2 1 8)
(f' 2 0 16)
16
```

Notice, that the size of the expression *does not grow*, hence the function could in theory be rewritten as a simple loop with bounded storage space requirements, a concept more familiar from imperative programming languages such as C or C++. The final improvement that needs to be applied is actually telling the interpreter to perform tail call optimisation in this case, using the self-referential call `&0`:

```
--> defun power (x n) ((lambda (x n acc) (if (= n 0) acc (&0 x (- n 1) (* x acc)))) x n 1)
(λ x n . ((λ (x n acc) (if (= n 0) acc (&0/syn x (- n 1) (* x acc)))) x n 1))
--> power 2 4
16
```

## 1.5. Function composition

Function composition is a a common core concept in functional programming languages, an emphasis on which is placed in KamilaLisp. Using the `@` operator, it is possible to compose two or more functions into a single function. An example follows:

```
--> defun f x (* x 2)
(λ x . (* x 2))
--> defun g x (+ x 1)
(λ x . (+ x 1))
--> f 2
4
--> g 2
3
--> f@g 2
6
```

The usefulness of function composition might be hard to appreciate at first. In the end, `f@g x` ⇔ `f (g x)`, however, the function returned by the `@` operator does not have to be immediately applied - it can be bound to a name and used later. Without the composition operator, it would be necessary to introduce a *lambda* to achieve the same result as `f@g` - that is, `lambda x (f (g x))`. The difference between manually composing two functions to create a new function and using the `@` operator is how they treat arguments. The `@` operator variant of the same expression does not refer to the arguments of the composed functions (and thus does not manually relay an indeterminate amount of them to the innermost function), creating the basic building block for *point-free programming*.

Of course, it is possible to compose arbitrarily many functions by using the `@` operator multiple times. The following example demonstrates the composition of three functions:

```
--> defun f x (* x 2)
(λ x . (* x 2))
--> defun g x (+ x 1)
(λ x . (+ x 1))
--> defun h x (- x 3)
```

```
(λ x . (- x 3))
--> f@g@h 2
0
```

Another form of function composition notoriously used in the APL family of programming languages is the so-called *fork* (the $\mu$-recursive composition operator). Generally speaking, it is sometimes of special interest to preprocess the arguments using different functions (*reductees*), and then funnel the results into a single function (*reductor*). For instance, the arithmetic mean is defined as the quotient (*reductor*) of the sum and length of a list (*reductees*), while a palindrome is defined as a string for which the equality (*reductor*) between it and its reverse (*reductees*).

To implement a sum function using a fork, it is necessary to define a function that sums a list beforehand using the `tally` function which returns the length of a list. A non-zero value is considered truthy, hence the following definition of the *sum* function:

```
--> defun sum (l) (if (tally l) (+ (car l) (sum (cdr l))) 0)
(λ l . (if (tally l) (+ (car l) (sum (cdr l))) 0))
--> sum '(1 6 2 3)
12
```

There are many alternative ways to implement it. One problem the reader may stumble upon is `= 'nil 'nil` returning `'nil`. This behaviour is reasoned by the fact that equality *vectorises* over lists, meaning that equality is tested element-wise. To determine structural equality, it is advised to use the `same` function. The following example demonstrates this behaviour:

```
--> = '(1 2 3) '(1 2 4)
(1 1 0)
--> same '(1 2 3) '(1 2 4)
0
--> same '(1 2 3) '(1 2 3)
0
```

Hence arguing for a simpler implementation of the *sum* function:

```
--> defun sum (l) (if (same l '()) 0 (+ (car l) (sum (cdr l))))
(λ l . (if (same l '()) 0 (+ (car l) (sum (cdr l)))))
--> sum '(1 6 2 3)
12
```

The arithmetic mean function is thusly introduced as follows:

```
--> [/ sum tally] '(1 6 2 3 4)
3.2
```

Of course, it could also be bound to a name without applying the fork instantaneously:

```
--> def avg [/ sum tally]
[/ sum tally]
--> avg '(1 6 2 3 4)
3.2
```

The key merits of point-free programming are:

- The code is more concise.

- The code is often more readable.
- The code is faster, because there is no need to allocate a stack frame.
- The code does not need to bind any names.

The lack of argument naming gives the point-free paradigm a reputation of being unnecessarily obscure, hence the humorously used epithet "pointless style"[1]. However, the point-free paradigm is not necessarily obscure. In fact, it is often more readable than the equivalent imperative code and favoured by many languages, such as Haskell, APL, J, PostScript, Forth, Factor, jq and the Unix shell[2].

## 1.6. Partial application and $\mu$-recursive functions

Most functional programming languages, such as OCaml and Haskell automatically *curry* functions by default, that is, allow applying functions to fewer arguments than they are defined to take. This is a useful feature, because it allows for the creation of new functions by binding some of the arguments of a function to a value. This mechanism is called *partial application*. KamilaLisp supports partial application of functions but it does not happen by default, since unlike OCaml and Haskell, KamilaLisp supports variadic functions. Recall the following example given earlier in the book:

```
--> ; Returns a function that adds a given number to its argument.
--> ; The technique demonstrated is often called "currying".
--> defun make-adder (x) (lambda y (+ x y))
(λ x . (λ y . (+ x y)))
--> def add-5 (make-adder 5)
(λ y . (+ 5 y))
--> add-5 10
15
```

Using partial application, the example can be rewritten as follows:

```
--> defun add-x n $(+ n)
(λ n . $(+ n))
--> def add-5 (add-x 5)
$(+ n)
--> add-5 6
11
```

Partial application is particularly useful in conjunction with various kinds of function composition, as it allows for the vast majority of functions that are defined in terms of other functions to be defined in a point-free manner.

Variadic functions were briefly mentioned earlier in the book. Simply speaking, they are functions that take an arbitrary number of arguments. `+` is a good example of a variadic function, since when applied to more than two arguments, it just sums everything:

```
--> + 1 2 3 4 5
15
```

[1] http://hdl.handle.net/1822/2869

[2] The pipe operator is essentially point-free function composition. The Unix shell has more interesting properties from a category theoretic perspective, due to its extensive use of monads. The pipe operator is an implementation of `bind`, `cat` is in reality monadic `return`. Moreover, a pair of these operations, adding `<` and `>`, satisfy the monadic laws - `cat f | cmd` is the same as `cmd <f` (left-hand identity); `cmd | cat` is the same as `cmd` (right-hand identity); `c1 | (c2 | c3)` is the same as `(c1 | c2) | c3`.

It is possible to define custom variadic functions in KamilaLisp using special argument syntax. To recall, `lambda (x y) (code)` takes two arguments - $x$ and $y$. This function can be made variadic by prepending the *last argument* with an ellipsis - `lambda (x ...xs) (code)`. The last argument is bound to a list of all the remaining arguments. This has an interesting property: while the aforementioned function takes *at least* one argument (because `...xs` can be empty), it can be modified to take any amount of arguments, even zero - `lambda ...xs (code)`. To demonstrate this behaviour, the following function will obtain the arithmetic average of all its arguments, but will refuse to be called with zero arguments:

```
--> defun avg (x ...xs) (/ (+ x (sum ...xs)) (+ 1 (tally ...xs)))
(λ x ...xs . (/ (+ x (sum ...xs)) (+ 1 (tally ...xs))))
--> avg 1 6 2 3 4
3.2
--> avg 5
5
--> avg
TypeError thrown in thread 15eb5ee5:
        Expected at least 1 arguments to `(λ x ...xs . (/ (+ x (sum ...xs)) ...))'.
    at entity avg  1:1
    at (λ x ...xs . (/ (+ x (sum ...xs)) ...))  1:11
```

Equipped with the power of variadic functions and recursion, the next natural step is to define $\mu$-recursive functions. Consider the following *basic* $\mu$-recursive functions:

- For all natural numbers $i, k$ where $0 \leq i \leq k$, the projection function (sometimes also called the identity function when $i = 0$) is defined as $P_i^k(x_0, \ldots, x_k) = x_i$. The projection function is a built-in operator in KamilaLisp, where `#a` is equivalent to $P_a^k$.
- For each natural $n$ and $k$, the constant function $C_n^k$ is defined as $C_n^k(x_0, \ldots, x_k) = x_n$. This is easily implemented using the previously discussed projection function as `$(#0 n)`.
- For each natural $n$, the successor function $S_n$ is defined as $S_n(x) = x + 1$. This is easily implemented using just function composition - `$(+ 1)`.

The $\mu$-recursive composition operator (also called the substitution operator) defined for an $m$-ary function $h(x_0, \ldots, x_m)$ and exactly $m$ $n$-ary functions $g_0(x_0, \ldots, x_n), \ldots, g_m(x_0, \ldots, x_n)$ as $h \circ (g_0, \ldots, g_m) = f$ where $f(x_0, \ldots, x_n) = h(g_0(x_0, \ldots, x_n), \ldots, g_m(x_0, \ldots, x_n))$. This definition is essentially equivalent to KamilaLisp *forks*[3] - `[h g_0 ... g_m] x_0 ... x_k`.

The $\mu$-recursive primitive recursion operator $\rho(g, h) = f$ for $k$-ary function $g(x_0, \ldots, x_k)$, $k+2$-ary function $h(y, z, x_0, \ldots, x_k)$ and $k+1$-ary function $f$ yields the following piecewise function:

$$f(a, x_0, \ldots, x_k) = \begin{cases} g(x_0, \ldots, x_k) & \text{if } x = 0 \\ h(a-1, f(a, x_0, \ldots, x_k), x_0, \ldots, x_k) & \text{if } x \neq 0 \end{cases}$$

The KamilaLisp implementation of this concept is slightly more involved, requiring the `lift` function to apply a function on a existing variadic parameter pack:

```
--> defun mu-prim-rec (g h) (
...     lambda (a ...xs) (
...         if (= a 0)
...             (lift g ...xs)
...             (lift $(h (- a 1) (lift $(mu-prim-rec g h) ...xs)) ...xs)))
```

[3] A general version of APL 3-trains; https://aplwiki.com/wiki/Train#3-trains

The $\mu$-recursive *minimization* operator is less demanding to implement. Intuitively, minimisation seeks, beginning the search from 0 and proceeding upwards, the smallest argument that causes the function to return zero; if there is no such argument, of if one encounters an argument for which f is not defined, then the search never terminates, and is not defined for the argument.

```
--> defun mu-min f (lambda ...xs (let-seq (
...    (defun mu-min-iter i (if (lift f ...xs) (&0 (S i)) i))
...    (mu-min-iter 0))))
```

## 1.7. Iteration

Given a function $f$, the $n$-fold application of $f$ to $x$ is usually denoted in mathematics as $f^n(x)$. For instance, $f^0(x) = x$, $f^1(x) = f(x)$, and $f^2(x) = f(f(x))$. More generally, a recursive relation can be defined as $f^k(x) = f(f^k - 1(x))$ for $k \geq 1$. Given *an iteration* of a function $f^n(x)$, the function $f$ is called the *step function*, $n$ is called the order of iteration and $x$ is called the *starting value*. In KamilaLisp, iteration is introduced using the `while` higher order function which, in the presently most useful form to the mathematical definition, takes three arguments - the starting value, the order of iteration and the step function.

Consider the successor function `defun s x (+ x 1)`. To provide a basic example, addition of two numbers `+ a b` can be implemented as an iteration of order $b$ of the successor function `s` with the initial value $a$:

```
--> defun add (a b) (while a b $(+ 1))
(λ a b . (while a b $(+ 1)))
--> add 3 4
7
```

Recall the power function and its previously discussed, tail-recursive definition:

```
--> defun power (x n) ((lambda (x n acc) (if (= n 0) acc (&0 x (- n 1) (* x acc)))) x n 1)
(λ x n . ((λ (x n acc) (if (= n 0) acc (&0/syn x (- n 1) (* x acc)))) x n 1))
```

Since tail recursion is essentially equivalent to iteration, the power function can be implemented as an iteration of order $n - 1$ of the multiplication function `*` with the initial value $x$:

```
--> defun power (x n) (while x (- n 1) $(* x))
(λ x n . (while x (- n 1) $(* x)))
--> power 2 4
16
```

Iteration using `while` and other higher order functions is described later on in the book. As a tool, iteration is more suitable to certain problems over recursive (functional) or array programming approaches[4].

## 1.8. Exceptions

[4]For instance, computing the *Convex Hull* using the Graham scan algorithm, which has better asymptotic complexity than the more naive Jarvis scan as easily implemented in array fashion

# Chapter 2

# List processing